Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services

Chaoshun Zuo
University of Texas at Dallas
cxz153430@utdallas.edu

Wubing Wang
University of Texas at Dallas
wxw132530@utdallas.edu

Rui Wang
AppBugs, Inc
rui@appbugs.co

Zhiqiang Lin
University of Texas at Dallas
zxl111930@utdallas.edu

***Abstract*—Most mobile apps today require access to remote services, and many of them also require users to be authenticated in order to use their services. To ensure the security between the client app and the remote service, app developers often use cryptographic mechanisms such as encryption (e.g., HTTPS), hashing (e.g., MD5, SHA1), and signing (e.g., HMAC) to ensure the confidentiality and integrity of the network messages. However, these cryptographic mechanisms can only protect the communication security, and server-side checks are still needed because malicious clients owned by attackers can generate any messages they wish. As a result, incorrect or missing server side checks can lead to severe security vulnerabilities including *password brute-forcing*, *leaked password probing*, and *security access token hijacking*. To demonstrate such a threat, we present AUTOFORGE, a tool that can automatically forge valid request messages from the client side to test whether the server side of an app has ensured the security of user accounts with sufficient checks. To enable these security tests, a fundamental challenge lies in how to forge a valid cryptographically consistent message such that it can be consumed by the server. We have addressed this challenge with a set of systematic techniques, and applied them to test the server side implementation of 76 popular mobile apps (each of which has over 1,000,000 installs). Our experimental results show that among these apps, 65 (86%) of their servers are vulnerable to password brute-forcing attacks, all (100%) are vulnerable to leaked password probing attacks, and 9 (12%) are vulnerable to Facebook access token hijacking attacks.**

I. INTRODUCTION

Today mobile apps are everywhere. They range from simple information gathering applications, such as for retrieving email, news, and weather, to feature rich applications, such as for mobile gaming, online banking/shopping, and blogging/chatting. In Google Play, which is one of the most popular app stores, there are over 1.6 million Android apps in total, with more than 50 billion downloads [4]. Meanwhile, the popularity of mobile apps has continued to rise due to their increasingly prevalent usage across mobile device (e.g., smartphone and tablet) users.

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23146

To save client storage and energy consumption, there is usually a remote party involved in mobile computing. Specifically, similar to the traditional desktop web-browser based computing, a mobile app also often needs to interact with a remote service, e.g., to retrieve the data of a user's interest such as the weather information where the user lives. To provide customized services and also prevent resource abuse, a typical step to get the access is through user authentication. Therefore, many mobile apps today require users to register with the service providers first, and then use their services after authentication.

As a result, it is crucial to ensure the security of the authentication process. There are various ways that mobile app developers have used over the years to achieve this. For instance, they can encrypt the traffic between the mobile app and the server (e.g., through HTTPS), they can hash (e.g., through MD5, SHA1) the user password before sending to the server for authentication, and they can also sign (e.g., through HMAC) each message generated from the mobile app. Correspondingly, on the server side, the server needs to decrypt each message, validate the hash or the signature of the message, and reject all the invalid ones.

While it appears to be secure if the server rejects all of the invalid messages, such security is based on the assumption that a client cannot forge a valid message. Unfortunately, in this paper we show that such an assumption is false, and a client can completely break the message authentication including cryptographic hashing and signing and generate "legal" messages for the server to consume. This is because an attacker can completely control a client app (e.g., running in an emulator), analyze (i.e., reverse engineer) how a valid message is generated, and correspondingly generate forged messages.

Consequently, in addition to message decryption, hashing and signature checking, the server also needs to perform additional security checks. Otherwise, this can lead to a number of security vulnerabilities. One such vulnerability is password brute-forcing. In particular, if the server does not maintain the state of how many passwords a user has tried while attempting to login within a certain time window, an attacker would be able to figure out the user's password by continuously guessing it. Also, being able to forge valid request messages would allow attackers to probe the existence of certain users using leaked usernames and passwords (due to the common practice of password reuse among many users [15], [21]). Meanwhile, the lack of a server side security check can also lead to an access token hijacking attack [2], [36]. Specifically, an attacker can forge a valid message by

using a stolen token from other apps to bypass the server side authentication of the target app (if the server is vulnerable) and then use the target app's service. In addition, there could also exist a SQL injection attack if the server does not perform the sanitation check of the "legal" messages from the client since an attacker is now able to forge any messages.

To demonstrate the threat of these security vulnerabilities at the server side, this paper presents AUTOFORGE, a tool that can automatically forge cryptographically consistent messages for the security testing of mobile services when given a mobile app. It contains a set of black-box techniques including API hooking, lightweight protocol field reverse engineering, and request message forgery to automatically generate valid request messages. At a high level, AUTOFORGE works as follows: given an app and a few legal inputs (e.g., a username with a correct and wrong password), it observes how the user input is processed by only hooking a set of well known cryptographic APIs, and intercepts the outgoing messages with a man-in-the-middle network proxy; next, it infers the message fields and their semantics by diffing the messages and measuring the degree of the differences; after that, it forges the messages by only mutating the protocol fields of interest (e.g., username and password) and generating the cryptographically computed fields through an out-of-box re-execution (i.e., replay) of the cryptographic APIs.

We have implemented AUTOFORGE, and tested with 76 popular mobile services by running the corresponding mobile apps. One criteria for selecting which service to test is based on whether the client apps have been installed over one million times. We have obtained very encouraging experimental results. Among the 76 tested services, we found that 65 (86%) servers (including CNN, Expedia, iHeartRadio, and Walmart) are vulnerable to password brute-forcing attacks, all (100%) of them are vulnerable to leaked password probing attacks, and 9 (12%) of them are vulnerable to Facebook access token hijacking attacks.

In short, we make the following contributions:

- We show that the server side implementation of many mobile apps lacks sufficient security checks and is vulnerable to a number of malicious login attacks including password brute-forcing, leaked password probing, and access token hijacking.
- We present a set of lightweight techniques to automatically forge cryptographically consistent messages. Our technique does not require sophisticated reverse engineering of the mobile apps, and instead by only hooking a set of well known cryptographic APIs and using a lighweight protocol reverse engineering with an out-of-the-box re-execution of the cryptographic functions we successfully forge valid request messages.
- We have implemented our techniques in AUTOFORGE, and applied it to test 76 popular mobile apps (each has over one million installs), and we have found that the majority of these app servers are vulnerable to malicious login attempts. We have made responsible disclosure and notified each vulnerable app vendor, and three of them have patched their service shortly after our notification.

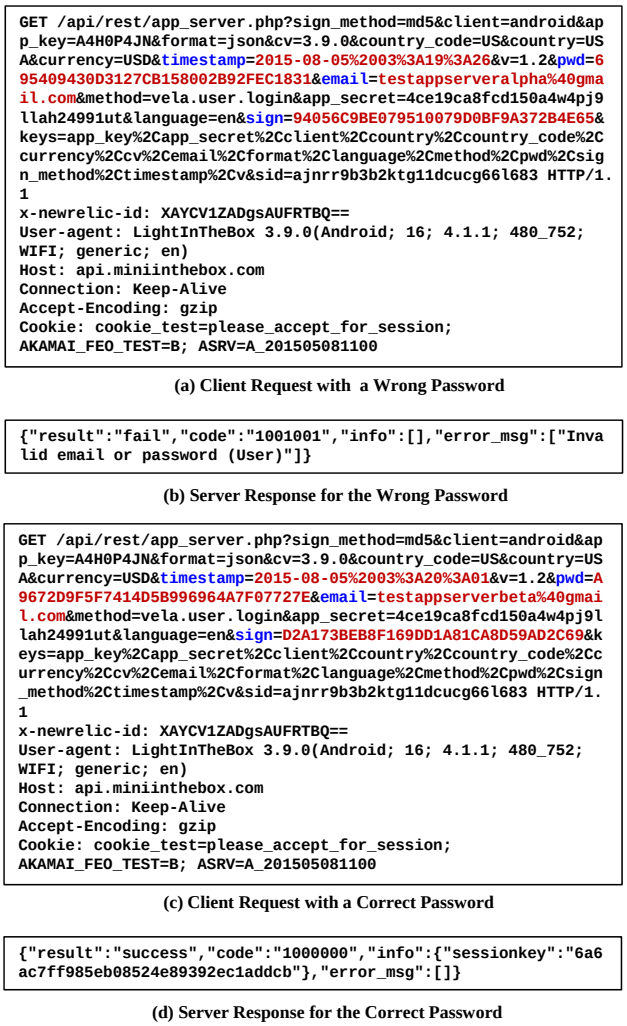

Fig. 1. Network Traces of the Login Attempts of `miniinthebox` App.

II. BACKGROUND AND OVERVIEW

The goal of this paper is to develop techniques that can automatically forge valid cryptographically consistent client request messages, and apply them to find the security vulnerabilities (such as password brute-forcing) in the server side. In this section, we provide the necessary background and give an overview of how we achieve this goal. We first start from a running example (§II-A) to illustrate the challenges and present our observation (§II-B), and then we define our research problem and overview our system (§II-C).

A. A Running Example

To understand our problem better, Fig. 1 illustrates the network traces gathered from the popular Android app `miniinthebox`. It is an online shopping app which has one-to-five million installs according to Google Play. As shown in Fig. 1, we performed two tests: the first is to enter a wrong password (`1234567890`) for user `testappserveralpha@gmail.com`, and the client request message and the server response message are illustrated in Fig. 1(a) and (b); the other is to enter a correct password (`ThisIsPWD!`) for a different user, `testappserverbeta@gmail.com`, whose request and

response messages are illustrated in Fig. 1(c) and (d), respectively. We can notice from the trace that this app uses the plain-text HTTP protocol, and there are many app-defined protocol fields in this login request message such as `sign_method`, `client`, `app_key`, `format`, `pwd`, `email`, `sign`, `keys`, and `sid`, etc.

Among these protocol fields, a few of them are of special interest to us such as `pwd`, `email`, and `sign` if we aim to perform a password guessing test. That is, we can keep mutating a user password (from `1234567890` to some other dictionary guided guesses) and test whether the server accepts or rejects our password. However, we can notice that the user entered password `1234567890` has been hashed (or encrypted) to value `695409430D3127CB158002B92FEC1831`. Meanwhile, there is a `sign` field that is a cryptographic signature of the client request message, and the server will verify whether the `sign` field is correct or not. Also, we can notice that the value of the `sign` field is significantly different in the two request messages.

Therefore, in order to generate valid request messages, we just need to recognize the message fields of interest to us such as the `pwd` and `sign` field, mutate the corresponding field (e.g., the `pwd`), and generate valid cryptographically consistent fields (e.g., `sign`) of the request message. In addition, we also need to monitor the response of the server packets, to terminate the test once we find a correct password.

B. Observation

Challenges. From our running example, we can notice that there are a number of challenges in order to perform server side security testing:

- **Recognizing the protocol fields**. Typically a network message consists of a number of fields; some of them are standard protocol fields (e.g., `GET`), while some are user defined. While it might be easier to identify the standard fields for well-known protocols, it will be much more challenging to recognize the user defined fields, especially considering the fact that different developers can name a field differently (e.g., they might use either `pwd`, `passwd`, or `password` for a password field).

- **Identifying the cryptographic functions**. To encrypt or hash a password, different apps can also use different cryptographic functions (e.g., `MD5`, `SHA-1`, `AES`, `DES`, etc.). Similarly, to generate the signature of a protocol message, apps can also use different message authentication code (MAC) generation functions (e.g., `HMAC`, `HMAC-SHA-1`). We need to identify the functions that are used by the testing app, so as to regenerate the corresponding password, hash, or signature. Meanwhile, an app might use their own private cryptographic functions, though this is not encouraged.

- **Deciding when to terminate**. We cannot perform a brute-force test forever, and we must terminate at some point. While it might appear to be very simple by parsing the response messages from the server (e.g., by looking at the `success` or `fail` string as shown in Figure 1 (b) and (d)), such an approach would be too app-specific since different apps can use different strings and different encoding to represent a succeeded or failed attempt.

- **Generating the valid request messages**. Having recognized the message fields of our interest, we also have to finally generate the new valid messages for our testing. While it might be possible to dynamically instrument the app and use an in-context argument substitution of the cryptographic APIs to generate the message, or just fuzz the graphic user interface to generate the "legal" messages, these approaches appear to be more expensive or lack flexibility (e.g., requiring recognizing and controlling of the user interface, rolling back the state of the login event, or only substituting user visible fields) and instead we would like to have an out-of-the-box approach to forge any "legal" messages as we wish.

Key Insights and Solutions. At a high level, we can notice that essentially we are performing protocol reverse engineering in that we have to recognize the protocol fields, understand the request and response messages (to a certain degree), and generate valid messages with cryptographically computed fields. While we could adopt many of the existing protocol reverse engineering techniques (e.g., [10], [14], [22], [25], [39]) to analyze at the instruction level how a message is generated, such an approach also appears to be more expensive since it tracks the data dependency at the instruction level. Having analyzed the executions of a number of apps manually, we have obtained the following insights to address those technical challenges discussed above:

- **Inferring the message fields with diffed input**. Although it is challenging to recognize each field in a given message, we realize that we need to infer only a few of them based on our interests (e.g., only the `email`, `pwd`, and `sign` fields in our running example). Since we control the app execution, we can feed the app with controlled input such as a correct password and a wrong password. By observing the request message differences, we can identify the diffed fields. The fields of our interest must be within the diffed fields. For instance, as shown in Fig. 1(a) and (c), there are only four diffed fields: `timestamp`, `pwd`, `email`, and `sign`, and we can quickly narrow them down by using request message diffing.

- **Dynamically hooking well-known cryptographic APIs**. While an app can use different types of cryptographic functions for encryption, hashing and signing of a message, there are only a limited number of them. Meanwhile, even though there might be some user defined cryptographic functions, these apps would be rare because of the "never-implement-your-own-crypto" practice [30]. Therefore, we can dynamically hook the well-known cryptographic APIs used by an app, extract their arguments (usually the user typed input such as the password or the fields that need to be digitally digested or signed will appear in the arguments) and return values that allow us to change only the arguments of our interest. Then, we can replay the execution of the cryptographic APIs with the new arguments to re-generate new valid messages.

- **Labeling response message with controlled input**. Similar to how we infer the message fields through diffed input, we can also infer the type of the response

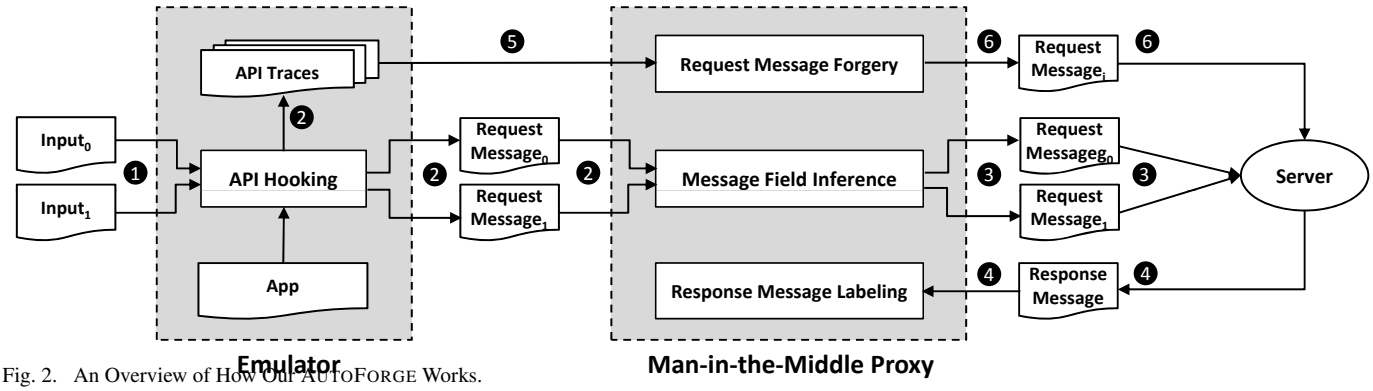

Fig. 2. An Overview of How Our AUTOFORGE Works.

message (namely, the success or failure login messages sent by the server) with controlled input diffing. More specifically, since we control the app, we can test the app with a correct password and treat the response message as a black box without looking at any of its content by assigning it a *success* tag; similarly, we can send a wrong password, and assign a *failure* tag for the corresponding response message. There will be some other types of messages, such as a too-many-login-attempts warning message sent from the server, but we can just assign all of these messages with an *other* tag regardless of their contents.

- **Out-of-the-box re-execution of the cryptographic functions**. An interesting observation for cryptographic function is that their algorithms are well-known, and different implementations by different programming languages such as Java, C, or Python would produce the same output when given the same input. Therefore, we can perform an out-of-the-box re-execution of the cryptographic functions to forge the desired request messages by feeding them with the corresponding arguments.

C. Overview

Problem Statement. After describing the challenges and our observations, next we would like to formally define our problem. It can be summarized as follows: *Given an app and traced input messages, the goal of* AUTOFORGE *is to automatically generate a new input message with mutated fields that satisfy the cryptographic constraints of the messages in an efficient and black-box manner.*

Scope and Assumptions. In this paper, we focus on testing the mobile services of Android apps. As to-be-demonstrated, we only need the knowledge of publicly available cryptographic APIs (e.g., the parameters and return values) as well as the capability of hooking these functions, and we assume these information is available. In addition, since our goal is to generate valid client side request messages, we need to reverse engineer the protocol fields. In this paper, we focus on the apps that use text-based protocols including HTTP/HTTPS because we can directly identify the protocol fields based on text differences.

Interestingly, many mobile apps in Android do use HTTP/HTTPS protocols, which makes it trivial in identifying the protocol fields through input message diffing. Note that for HTTPS, we can intercept their traffic and decrypt it by using a man-in-the-middle proxy. This is because we can easily install a self-signed root certificate in our testing Android device, and intercept and decrypt the traffic in a network proxy.

Overview. We have designed a set of systematic techniques in our prototype AUTOFORGE. As illustrated in Fig. 2, there are four key components inside AUTOFORGE: one is located inside an Android emulator, and the other three are located inside a man-in-the-middle (MitM) proxy. There are in total six major steps in order to forge a cryptographically consistent request message:

- **Step ❶**. To test a given app, we first need to provide the necessary input that generates the desired message fields. For instance, to test whether a service is vulnerable to password brute-forcing attack, we need to enter two testing inputs[1]: a testing username with the correct password for this user, and a testing username with a wrong password for this user, respectively. To have the correct password, we need to register with the service first. Therefore, **Step ❶** is the only manual step that involves human intervention. All other steps in AUTOFORGE are automatically executed.

- **Step ❷**. Once the app gets loaded and the input is fed to the app, our first component, *API Hooking*, will interpose the white-listed cryptographic APIs. Whenever one of the APIs is executed, we retrieve its input and output of this API from its arguments and return values based on the specification of the API. Such information is saved in a trace log. Later we will traverse the log file to generate the new request message in **Step ❺**. Meanwhile, the execution of the app inside our emulator will automatically generate a request message, which will be fed to our second component, *Message Field Inference*, and the copy of this message will also be sent to the server at **Step ❸** or right after the execution of **Step ❷**.

- **Step ❸**. By aligning the two request messages and diffing each message field, our *Message Field Inference* directly identifies the diffed message fields. Then it

[1] Strictly speaking, we need four inputs for the password brute-force testing. For space reasons we do not show them completely in Fig. 2. We will explain why we need four inputs in §III-C.

measures the similarity of the values between each diffed field. Based on the degree of differences, it identifies the cryptographically computed fields. A few other fields can also be inferred based on the pattern of the string (as we focus on text protocols), e.g. the `timestamp` field, which has a certain string inside such as the date of the test. The request message generated at **Step ❷** is sent to the server if it has not been sent yet. Note that the execution of **Step ❸** can be performed offline, and the system does not need to wait until this step is finished to execute **Step ❹**.

- **Step ❹**. The server sends a response message to the client, which is intercepted by our third component, *Response Message Labeling*. Based on the type of the message (e.g., the correct password, or a wrong password) we sent to the server, it assigns a corresponding label (or tag) to the response message (e.g., a `success` tag or a `failure` tag). We will also compare the tag for all later response messages (generated after **Step ❻**) to decide whether we should continue executing **Step ❻** based on the nature of the security testing we perform (e.g., repeatedly guessing a password until we get a `success` response).

- **Step ❺**. Having assigned the tag for the two initial response messages, and meanwhile having collected the input and output traces for each of the executed cryptographic APIs, our last component, *Request Message Forgery*, re-executes these executed cryptographic functions with the mutated input and finally generates the valid request message by replacing the corresponding field in the initial request message.

- **Step ❻**. The newly generated request message is sent to the server, and its response will be intercepted by our MitM proxy. Then we continue the execution to **Step ❹**.

III. Detailed Design

In this section, we present the detailed design of the four key components of AutoForge, based on the order of their execution.

A. API Hooking

The first component of AutoForge hooks the well-defined cryptographic functions to intercept their arguments and return values such that we can replay their execution to produce the desired cryptographically consistent fields. The Android SDK provides a set of cryptographic Java APIs. Based on their specification as well as our manual analysis with a number of apps, we have obtained 61 commonly used cryptographic APIs. Their prototypes are presented in Table I. Most apps[2] directly use them to encrypt input data (with the `crypto.cipher` class), generate a hash (with the `security.MessageDigest` class) or sign the input by generating a message authentication code (i.e., with the `crypto.Mac` class). Based on our manual analysis with a number of apps, we find these APIs are usually used in the following way:

- **Encryption**. To encrypt a message, an Android app first needs to initialize a cryptographic key class (e.g., by calling `new DesKeySpec` and `SecretKeyFactory.getInstance` to generate the DES keys), and then it calls `cipher.getInstance` with parameters such as "`DES/CBC/PKCS5Padding`" to get a `cipher` instance, and then `init` this `cipher` with the necessary parameters (e.g., the initialized keys). Then, app developers have to give the input message (using a byte array) to this `cipher` for encryption. There are two ways to do that: the first is to call API `doFinal` to pass the input and get output as cipher text; the second way is to call API `update` to pass the input, and then call API `doFinal` to produce the cipher text.

- **Hashing**. Obtaining a digest of a message (without using any keys) is achieved by using `MessageDigest` (e.g., `md5`, or `sha1`). In this case, the app calls `MessageDigest.getInstance` with string "`MD5`" as argument to get a MD5 MessageDigest instance, and then it calls the `update` method to add the message that needs to be digested. Finally, it calls `digest` to produce the desired hashing result.

- **Signing**. To sign a message (ensuring both integrity and authenticity), a message authentication code (i.e., `Mac`) is used. Similar to encryption, the app also has to generate the corresponding keys first (e.g., by calling `new SecretKeySpec` with string "`HmacSHA1`"), get a `Mac` instance by calling `Mac.getInstance` with a string (e.g., "`HmacSHA1`"), and then initialize the `Mac` with the generated key. Next, it calls `doFinal`, which takes the to be hashed messages as input and finally produces the hashed messages as output. It could also first call `update` to add the message, and then call `doFinal` with an empty argument.

Therefore, we hook each of the APIs (the handler and the function name) described in Table I, and log their arguments and return values. We log the arguments of these APIs right before their execution, and their return values as well as updated arguments if there are any right after their execution. A sample of our log is presented in Fig. 3.

B. Message Field Inference

Next, we need to identify the protocol fields of our interest in the request message. We divide this problem into two sub-problems: (1) message field identification that splits the messages into a set of fields, and (2) field semantic inference that infers the meaning of the identified fields. The outcome of this step is the fields we aim to mutate, such as `pwd` and `sign` in our running example.

1) Message Field Identification. Since we only need to substitute a few fields in our security testing, there is no need to identify all protocol fields. In addition, since we control the input to the testing app, we can observe the field differences in the request messages if we feed different inputs to the app. Based on these two insights, we can identify the fields that get changed by aligning the two request messages that are generated with

[2]There are apps that use native code and we need to hook the native code APIs in this case.

TABLE I. THE LIST OF THE HOOKED CRYPTOGRAPHIC APIS, AND ITS PARAMETERS AND RETURN VALUES.

Return Value	API name	Parameters
SecretKeySpec	javax.crypto.spec.SecretKeySpec.SecretKeySpec<init>	(byte[] key, String algorithm)
SecretKeySpec	javax.crypto.spec.SecretKeySpec.SecretKeySpec<init>	(byte[] key, int offset, int len, String algorithm)
DESedeKeySpec	javax.crypto.spec.DESedeKeySpec.DESedeKeySpec<init>	(byte[] key)
DESedeKeySpec	javax.crypto.spec.DESedeKeySpec.DESedeKeySpec<init>	(byte[] key, int offset)
DESKeySpec	javax.crypto.spec.DESKeySpec.DESKeySpec<init>	(byte[] key)
DESKeySpec	javax.crypto.spec.DESKeySpec.DESKeySpec<init>	(byte[] key, int offset)
X509EncodedKeySpec	java.security.spec.X509EncodedKeySpec<init>	(byte[])
SecretKeyFactory	javax.crypto.SecretKeyFactory.getInstance	(String algorithm)
SecretKeyFactory	javax.crypto.SecretKeyFactory.getInstance	(String algorithm, String provider)
SecretKeyFactory	javax.crypto.SecretKeyFactory.getInstance	(String algorithm, Provider provider)
SecretKey	javax.crypto.SecretKeyFactory.generateSecret	(KeySpec keySpec)
IvParameterSpec	javax.crypto.spec.IvParameterSpec.IvParameterSpec	(byte[] iv)
KeyFactory	java.security.KeyFactory.getInstance	(String algorithm)
KeyFactory	java.security.KeyFactory.getInstance	(String algorithm, String provider)
KeyFactory	java.security.KeyFactory.getInstance	(String algorithm, Provider provider)
PublicKey	java.security.KeyFactory.generatePublic	(KeySpec keySpec)
Mac	javax.crypto.Mac.getInstance	(String algorithm)
Mac	javax.crypto.Mac.getInstance	(String algorithm, String provider)
Mac	javax.crypto.Mac.getInstance	(String algorithm, Provider provider)
void	javax.crypto.Mac.init	(Key key)
void	javax.crypto.Mac.init	(Key key, AlgorithmParameterSpec params)
void	javax.crypto.Mac.update	(byte input)
void	javax.crypto.Mac.update	(byte[] input)
void	javax.crypto.Mac.update	(ByteBuffer input)
void	javax.crypto.Mac.update	(byte[] input, int offset, int len)
byte[]	javax.crypto.Mac.doFinal	()
byte[]	javax.crypto.Mac.doFinal	(byte[] input)
void	javax.crypto.Mac.doFinal	(byte[] output, int outOffset)
MessageDigest	java.security.MessageDigest.getInstance	(String algorithm)
MessageDigest	java.security.MessageDigest.getInstance	(String algorithm, String provider)
MessageDigest	java.security.MessageDigest.getInstance	(String algorithm, Provider provider)
void	java.security.MessageDigest.update	(byte input)
void	java.security.MessageDigest.update	(byte[] input)
void	java.security.MessageDigest.update	(ByteBuffer input)
void	java.security.MessageDigest.update	(byte[] input, int offset, int len)
byte[]	java.security.MessageDigest.digest	()
byte[]	java.security.MessageDigest.digest	(byte[] input)
int	java.security.MessageDigest.digest	(byte[] buf, int offset, int len)
Cipher	javax.crypto.Cipher.getInstance	(String transformation)
Cipher	javax.crypto.Cipher.getInstance	(String transformation, String provider)
Cipher	javax.crypto.Cipher.getInstance	(String transformation, Provider provider)
void	javax.crypto.Cipher.init	(int opmod,Key key)
void	javax.crypto.Cipher.init	(int opmod,Certificate certificate)
void	javax.crypto.Cipher.init	(int opmod,Key key,SecureRandom random)
void	javax.crypto.Cipher.init	(int opmod,Certificate certificate,SecureRandom random)
void	javax.crypto.Cipher.init	(int opmod,Key key,AlgorithmParameterSpec params)
void	javax.crypto.Cipher.init	(int opmod,Key key,AlgorithmParameterSpec params,SecureRandom random)
void	javax.crypto.Cipher.init	(int opmod,Key key,AlgorithmParameters params)
void	javax.crypto.Cipher.init	(int opmod,Key key,AlgorithmParameters params,SecureRandom random)
byte[]	javax.crypto.Cipher.update	(byte[] input)
byte[]	javax.crypto.Cipher.update	(byte[] input,int inputOffset,int inputLen)
int	javax.crypto.Cipher.update	(ByteBuffer input, ByteBuffer output)
int	javax.crypto.Cipher.update	(byte[] input,int inputOffset,int inputLen,byte[] output)
int	javax.crypto.Cipher.update	(byte[] input,int inputOffset,int inputLen,byte[] output,int outputOffset)
byte[]	javax.crypto.Cipher.doFinal	()
byte[]	javax.crypto.Cipher.doFinal	(byte[] input)
int	javax.crypto.Cipher.doFinal	(byte[] output, int outputOffset)
byte[]	javax.crypto.Cipher.doFinal	(byte[] input,int inputOffset,int inputLen)
int	javax.crypto.Cipher.doFinal	(byte[] input,int inputOffset,int inputLen,byte[] output)
int	javax.crypto.Cipher.doFinal	(byte[] input,int inputOffset,int inputLen,byte[] output,int outputOffset)
int	javax.crypto.Cipher.doFinal	(ByteBuffer input, ByteBuffer output)

the two controlled inputs. As shown in our running example, if we directly align (with a global optimal matching) the messages in Fig. 1(a) and (c), we immediately identify four fields (three are of special interest to us).

Then, the next question is how to find the two desired request messages for the alignment. A straightforward approach would be to align all request messages generated from the start of the app to the moment right after we trigger the login event. Presumably the two executions will share almost the same execution path except those code that handles input differences. While we can take such an approach, we realize that we can use a slightly better way to get the desired messages within only one execution of the app. In particular, after we load the app to test the login attempt, we can first enter a wrong password, and then enter a correct password. We would then just need to align the *two most recently generated request messages*. Though this is a heuristic approach, it works well in practice, and in our all testing apps we directly identify the two request messages desired for alignment.

After that, we compare the two request messages by using a pairwise string sequence alignment algorithm, namely the Needleman-Wunsch algorithm [27]. It uses dynamic programming and can achieve an optimal global matching, which perfectly fits our goal. Meanwhile, this algorithm has been used in the Protocol Informatics (PI) [8] project, and showed great promise for text based protocol field inference. Therefore, we just integrate this algorithm by following how PI uses it.

TABLE II. THE LEVENSHTEIN SIMILARITY RATIO OF THE DIFFED-FIELDS.

Field Name	$String_0$ vs. $String_1$	LSR
timestamp	2015-08-05%2003%3A19%3A26 2015-08-05%2003%3A20%3A01	0.84
email	testappserveralpha%40gmail.com testappserverbeta%40gmail.com	0.88
pwd	695409430D3127CB158002B92FEC1831 A9672D9F5F7414D5B996964A7F07727E	0.34
sign	94056C9BE079510079D0BF9A372B4E65 D2A173BEB8F169DD1A81CA8D59AD2C69	0.28

2) Field Semantic Inference. Having identified the diffed fields, we then infer their meanings. There are mainly three sources that lead to the field differences: (1) system data such as timestamp, (2) user input, and (3) the cryptographic computation. We present the following three strategies to infer their meanings:

- **Pattern Matching**. System data such as `timestamp` usually has patterns, and we can then use the pre-defined patterns to match them. For instance, if we locate a date sub-string such as `2015-08-05` in the two diffed fields, then it is highly likely that this is a `timestamp` field, as illustrated in our running example.
- **Content Matching**. Since we control the user input and some user input would not get changed, such as the username, then we directly search the diffed fields for the data we entered. In such a way, we can precisely locate the field that directly uses the user input, such as the `email` field in our running example.
- **Degree of Differences**. By measuring the degree of the similarities between the two diffed fields, we can easily identify the cryptographically computed fields. In our design, we use the Wagner-Fischer algorithm [35], which computes the Levenshtein distance, or minimum number of edits needed to transform one string into the other, between two fields. We determine whether a field is cryptographically computed if the Levenshtein similarity ratio (LSR) is below 0.5, as shown in Table II for our running example where we can easily locate the `pwd` and `sign` fields.

Note that *field semantic inference* is an optional step. In the worst case, AUTOFORGE can brute-force try each diffed field (e.g., there are only 4 fields in our running example that needs the brute-force trial) as crypto-field, system-fiel, or user-input field, to finally generate the desired request messages. With *field semantic inference*, the benefit is that it can significantly narrow down and even directly pinpoint the field of our interest.

C. Response Message Labeling

Since we aim to test the server behavior, we have to also monitor the server responses to decide when to stop. It would be very challenging to label a response message by parsing its contents since different apps can use different encodings. Fortunately, we find that we can actually treat the response message as a black box. Specifically, in our password login test, because the app is under our control, we can send the server two more messages in addition to the two initial request messages we sent earlier. Back to our running example, we have already collected two response messages: a wrong password response message (Fig. 1(b)), and a correct password response message (Fig. 1(d)). Then we can send another pair of messages, one with a wrong password and the other with the correct password, and use the following algorithm to label the response messages:

- If both the wrong (or correct) password response messages are content identical to the previously observed ones, then we directly use the corresponding entire message as a signature to classify whether it is a wrong (or correct) password response message.
- Otherwise, we align the two same type of response messages (i.e., two correct password response messages, or the two wrong password response messages) using again the Needleman-Wunsch algorithm [27], but we keep the common substring (instead of the diffed substring we used in **Step ❸**) and use it as a signature to represent a correct password response message or a wrong password response message.

After we have acquired the signatures for the correct password response and wrong password response, next we keep sending the server a login request with mutated passwords for a given user. However, for ethical reasons, we would not keep sending a large volume of mutated request messages to the server, and in our experiment we set the maximum number of messages we could send to the server as $N + 1$. During this testing window, we could observe three types of messages sent from the server:

- **Correct password**. We may break a user's password within $N + 1$ guesses, and the server will send a successful login response. Based on the already obtained signature of the correct password response, we identify this case.
- **Wrong password**. Given the very small amount of guesses, we likely cannot break a password. Therefore, most of the time, server will send a wrong password response message. Similarly to how we identify the correct password response message, we identify this case based on the already obtained wrong password signature.
- **Unrecognized response message**. In addition to the two correct or wrong password responses, we could also encounter other types of response messages that do not hold any signatures we observed before. The response for these messages could be something indicating we have exceeded a limited number of login attempts, or just an error message. Therefore, if we observe these unrecognized response messages, we terminate the test and conclude that the server is not vulnerable.

Note that there is also a caveat: if the server is not vulnerable, it may keep sending a wrong password response message even though we have guessed a correct password (in fact we did find two such servers in our experiment). Therefore, if we receive N wrong password responses, we will send a correct password for our testing user in the last request message. If the server blocks (by sending some other unrecognized response or the wrong password response), we conclude the server is not vulnerable.

Algorithm 1 Parsing the Cryptographic API trace and Tracking the Backward Data Dependency

```
1: Input: Log: the API execution log file; v_0: the value of the identified output field; u_0: user entered input;
2: procedure APITRACEPARSING(Log, v_0, u_0)
3:     V ← v_0
4:     H ← ∅, R ← ∅
5:     i ← 0
6:     while !feof(Log) do
7:         <handle.fname, input, output> ← fread(Log)
8:         API_i ← <handle.fname, input, output>
9:         i ← i + 1
10:    while i! = 0 do
11:        i ← i − 1
12:        if API_i.output ∈ V then
13:            V ← V \ {API_i.output}
14:            PUSHARGANDFUNNAME(API_i, V, H, u_0)
15:        if API_i.output ∈ H then
16:            H ← H \ {API_i.output}
17:            PUSHARGANDFUNNAME(API_i, V, H, u_0)
18:        if empty(V) and empty(H) then
19:            break
20:    if !empty(V) or !empty(H) then
21:        return false
22:    else
23:        return true
24: procedure PUSHARGANDFUNNAME(API, V, H, u_0)
25:    if String (API.input) then
26:        v_d ← GetDiffedArgValueFromTwoTraces()
27:        if u_o ∈ API.input or v_d ∈ API.input then
28:            PUSH (ARG, Substitute(v_d, u_0, API.input)))
29:            V ← V ∪ v_d
30:        else
31:            PUSH (ARG, String(API.input))
32:    else
33:        if CONST (API.input) then
34:            PUSH (ARG, CONST(API.input))
35:        else
36:            V ← V ∪ API.input
37:            PUSH (ARG-t, API.input.temp)
38:    if !empty(API.handle) then
39:        H ← H ∪ API.handle
40:    PUSH (FNAME, API.handle.fname)
```

D. Request Message Forgery

Having collected the API traces and identified the fields of our interest, we are then ready to forge the desired request messages for our security testing. For each diffed-field identified by our *Message Field Inference*, we substitute them either based on their inferred meaning or trying each of them one-by-one in a brute force way to forge a request message. The forgery of the request message is guided by the traced message as well as the traces of the cryptographic APIs. Since there are two types of fields, non-cryptographically computed fields and cryptographically computed fields, we use the following strategies to forge their values.

1) Non-cryptographically computed fields. For non-cryptographically computed *user input fields* such as `email` we forge the value of this field without changing its content (because we aim to test whether we can guess the password for a given user). For *system related fields*, such as `timestamp`, we configure AUTOFORGE to slightly change it based on the pattern observed in the traced request messages.

2) Cryptographically computed fields. The core problem AUTOFORGE aims to solve is to generate the cryptographically computed fields with mutated input. Once we have collected the traces of the cryptographic functions, including their input and output, all that we need to do is to replay the execution of these functions with the input we modified. Since our replay is performed at the network proxy layer, we just need to re-execute the cryptographic functions of our interest with the corresponding parameters. To identify those functions and their arguments, we perform backward slicing atop cryptographic API traces to identify the involved arguments and return values, and then replay their execution using the corresponding alternative (e.g., Python) implementation of these APIs. A detailed algorithm on how we parse the API trace and perform the slicing to identify the involved cryptographic functions is presented in Algorithm 1.

Specifically, given a log file of the API trace (LOG), the value v_0 of the identified cryptographically computed field (e.g., "`D2A173BEB8F169DD1A81CA8D59AD2C69`" in our running example), and the user input u_0 (e.g., "`ThisIsPWD`" and "`testappserverbeta@gmail.com`"), we invoke the APITRACEPARSING procedure to identify the functions that we need to replay along with the corresponding arguments. Since we start from the last executed API that generates the value of our interest and use the backward slicing to identify the replayed function, we use a stack structure (we call function state tracking stack) to store these functions and their arguments (as shown in line 27, line 30, line 33, and line 36 in Algorithm 1) and then we just need to pop these arguments and invoke the corresponding alternative implementation of these cryptographic APIs to finally produce the desired output.

Our backward slicing tracks two types of data dependencies: (1) function handler dependencies (stored in set H), and (2) return value and argument dependencies (stored in set V). As shown in line 12, starting from the return value of the last executed cryptographic API (e.g., the function `0x53595658.digest` illustrated in Fig. 3), if the return value belongs to V, then this function is of our interest; we therefore remove this return value from V (line 13) and push its argument and function name into our state tracking stack by calling procedure PUSHARGANDFUNNAME (line 14).

Inside PUSHARGANDFUNNAME procedure, we will first check its argument; if it is a string (line 25), then we again use the Needleman-Wunsch algorithm [27] to check whether its argument contains any diffed-value of our interest (e.g., `A9672D9F5F7414D5B996964A7F07727E` as shown in Fig. 3) by aligning the two corresponding arguments from the two traced API files, and storing the diffed value into v_d if there is any (line 26). Next, we further check if the user input u_0 (e.g., `testappserverbeta@gmail.com`) is in this argument, or if there is any diffed value v_d. If so, we will replace u_0 with either user specified input and meanwhile substitute the argument with a temporary variable that stores the v_d (line 28); we also track which function generates v_d by keeping it in V (line 29). Otherwise, we directly push this string argument (e.g., the "`DES`" string that is the argument of `SecretKeyFactory.getInstance` in Fig. 3) on the stack (line 31). If the argument is not a string (line 32-37), then we check whether it is a constant (e.g., the value `1` in `0x536b7670.init`'s argument). If so, we push this constant on the stack; otherwise, we will track which function generates this argument by adding it into data dependence set V, and push another temporary variable that will store the value generated by the dependent function. If the handler of this function is not empty (line 38), we track the dependence of the handler (line 39).

Fig. 3. Crypto API traces and the illustration of their arguments and return value dependencies of the `miniinthebox` App. Note that $*addr$ denotes the content stored in that $addr$.

Note that after we iterate the API traces, both V and H should be empty (line 20); otherwise there is something wrong and we will output that we cannot perform the replay.

After we have built the stack that tracks how the cryptographic functions should be executed, we then pop the arguments and the function names from the stack, and then invoke the corresponding alternative implementation of these cryptographic functions to finally generate the desired field output. After that, we replace the corresponding field in one of the request messages we traced (e.g, $\texttt{Request Message}_0$) to finally forge the desired request messages.

IV. Evaluation

We have implemented AutoForge using both Java and Python. We implemented our *API Hooking* in Java atop the `Xposed` Framework [6], which provides convenient ways to find and hook a given API (`findAndHookMethod`) and can intercept the point before (`beforeHookedMethod`) or after (`afterHookedMethod`) execution of the API. This implementation consists of $1,200$ lines of Java code. The rest of the components of AutoForge are implemented using Python with $4,500$ lines of our own code. It is worth noting that we implemented the *Message Field Inference* atop the Protocol Informatics [8] project, which is an open source Python implementation of the Needleman-Wunsch algorithm [27], and we just integrated this code based on our needs. Also, we did not have to implement the algorithm to compute the Levenshtein similarity ratio of two strings [35] because Python already has an implementation for this algorithm. Meanwhile, we implemented our MitM proxy atop the Burp Suite [1] using a Python plugin.

There will be many security applications enabled by AutoForge. In this section, we evaluate how we apply it to test the vulnerable app servers. In particular, we show how we tested whether an app server is vulnerable to password brute-forcing attacks in §IV-B, leaked username and password probing attacks in §IV-C, and the Facebook access token hijacking attack in §IV-D. Our procedure for setting up our experiments is presented in §IV-A.

A. Experiment Setup

Collecting the Mobile Apps for Testing. To test the app servers, we needed to first download and install the corresponding apps in our emulator. We crawled the apps from the official Google Play market. We crawled over $20,000$ apps within a three month time window. Since we have to manually register with each service in order to test whether their servers are vulnerable, we cannot test all of them and therefore we instead focused on the most popular apps. We considered an app to be a most popular app if it has been installed more than one million times. We queried each app to check its number of installs on Google Play; we found 320 apps falling into this category.

Among these 320 apps, not all of them use cryptographic functions to encrypt, hash, or sign the request messages, so we had to filter them. It would be tedious to manually go through each app one-by-one to check whether it uses cryptographic functions. We therefore developed a simple dynamic analysis tool based on Monkey [5] to decide whether we should filter an app. Specifically, we invoked the `am` command provided by Monkey to run the app and stop executing it after 20 seconds. If we observed any cryptographic functions (listed in Table I) get called, we kept this app for further testing.

After filtering the non-encryption, non-hashing and non-signing apps, we then had 105 apps to test. But still, we were not sure whether each app contained a user login interface since our test primarily concerns the security of user authentication. Currently, there is no automatic tool to recognize this, and therefore we had to go through each of them. After manually running the 105 apps one-by-one, we found that 15 of them do not contain a user login interface, and 14 of them do not use HTTP/HTTPS protocols. Therefore, we filtered these apps out and eventually had only 76 apps tested by AutoForge. The name of the tested app, its version, the category, and the number of installs, and the protocol (HTTP or HTTPS) are presented in Table V in Appendix. Also, we observed that 54 out of 76 (71%) apps in our data set use the HTTPS protocol.

Other Settings. We used Genymotion [3] as our Android emulator. Our host machine runs Ubuntu 12.04 with 8G memory and Intel Core2 Duo CPU 2.53GHz, and our Android emulator is version 4.2.2 with 2G memory. Meanwhile, the parameter N is set to be 20.

B. Password Brute-forcing Testing

We have illustrated through our running example how to break a user's password by iteratively mutating her password until we hit a correct one. We have applied this methodology to test these 76 potential vulnerable app services. To launch our test, we first registered two legal accounts in the corresponding servers and sent four request messages (a wrong and correct password pair for each registered user) and then mutating the password

TABLE III. THE DETAILED PASSWORD BRUTE-FORCING TESTING RESULT FOR 23 APP SERVERS BASED ON THE APP CATEGORY.

		Step ❶	Step ❷				Step ❸				Step ❹		Step ❺		
Category	**App Package Name**	**#Input Msg**	**#Traced API**	**Encryption?**	**Hashing?**	**Signing?**	**#DiffedField**	**#SysField**	**#InputField**	**#CryptoField**	**EqualResponse?**	**SysField Only?**	**#Sliced API**	**#Request**	**Vulnerable?**
Books & Reference	com.sirma.mobile.bible.android	4	146	✓	✗	✗	1	0	0	1	✗	✓	1	21	✓
Business	com.sahibinden	4	89	✓	✓	✗	4	1	2	1	✗	✓	15	21	✓
Casual	me.pou.app	4	169	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Comics	jp.ebookjapan.ebireader	4	60	✗	✓	✗	3	1	1	1	✗	✓	7	21	✓
Communication	com.browan.freeppmobile.android	4	40	✓	✓	✗	2	0	1	1	✗	✓	18	21	✓
Education	com.dictionary.flashcards	4	35	✗	✗	✓	5	2	2	1	✗	✓	9	21	✓
Entertainment	com.imdb.mobile	4	428	✗	✗	✓	4	1	2	1	✗	✓	7	21	✗
Finance	com.netgate	4	505	✓	✗	✗	3	1	0	2	✗	✓	28	6	✗
Health & Fitness	com.fatsecret.android	4	41	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Lifestyle	com.cookpad.android.activities	4	342	✗	✗	✓	4	1	2	1	✗	✓	1	21	✓
Media & Video	com.youku.phone	4	771	✗	✓	✗	4	1	1	2	✗	✓	7	5	✗
Medical	com.aranoah.healthkart.plus	4	321	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Music & Audio	com.slacker.radio	4	751	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
News & Magazines	com.cnn.mobile.android.phone	4	213	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Photography	com.picsart.studio	4	1292	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Productivity	com.autodesk.autocadws	4	153	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Shopping	com.biggu.shopsavvy	4	771	✗	✓	✗	3	0	2	1	✗	✓	8	21	✓
Social	com.tumblr	4	172	✗	✗	✓	5	2	2	1	✗	✓	7	21	✓
Sports	com.espn.score_center	4	385	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Tools	com.sohu.inputmethod.sogou	4	195	✗	✓	✗	2	0	1	1	✗	✓	7	3	✗
Transportation	taxi.android.client	4	35	✗	✓	✗	1	0	0	1	✗	✓	8	21	✓
Travel & Local	com.expedia.bookings	4	649	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Weather	disasterAlert.PDC	4	58	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓

for one of the registered legal users. It would be overwhelming to show all of the testing results for these 76 apps in a single table. We thus classify the apps based on their categories listed in Google Play, select the apps that have the highest number of installs in each category, and present their experimental results in Table III. In total, these apps can be classified into 23 categories. Therefore, there are only 23 app server testing results in Table III, and the results for the rest of the app servers are presented in Table VI in Appendix.

Specifically, we present the category of the app in the first column of Table III, followed by the app name. Since the execution of AUTOFORGE involves four key components, we present the internal results of these components in each key step from the 3rd column to the last column. In particular, the number of inputs needed in **Step ❶** is presented in the 3rd column. We can see that they all require 4 inputs. The 4th column reports how many APIs we traced, and the 5th to 7th column reports whether this app uses encryption, hashing, or signing, respectively, based on the execution of our *API Hooking* in **Step ❷**; The number of diffed fields by our *Message Field Inference* (**Step ❸**) is reported in the 8th column, and we also report the number of identified system data fields (e.g., the timestamp), user input data fields (e.g., username), and cryptographic computed fields from the 9th to the 11th columns. Whether our *Response Message Labeling* (**Step ❹**) observes identical response messages is reported in the 12th column; if they are not identical, whether the difference only comes from the system field is reported in the 13th column. Finally, we report the number of sliced APIs by our *Request Message Generation* (**Step ❺**) in the 14th column, the number of the request messages we sent in the 15th column, and whether the app server is vulnerable in the last column.

For these 23 apps' servers, we can observe from Table III that 19 (83%) are vulnerable to password brute force attacks with our limited 20 guesses. Note that if we also include the result (presented in Table VI) for the rest of the app servers, in total, we find 65 apps' servers (86%) are vulnerable to this attack type. Among the 4 non vulnerable apps servers in Table III, 3 of their servers (e.g., `com.netgate`) will directly return "Unrecognized response message" after 3, 5 or 6 request messages; but `com.imdb.mobile` will not return such message, and we only found it is not vulnerable after the 21st request message.From this table, we can also observe that we need four input messages for the test. Meanwhile, there are tens to several hundreds of cryptographic APIs executed for these tested apps. We have examined the traces and found that part of reason is because some of the apps heavily use cryptographic functions for integrity checking of the retrieved data such as the images before login. There are 65% of the apps that use encryption, hashing, or signing to protect the authentication request message; 17% use encryption, 39% use hashing, and 17% use signing. There are 8 apps (35%) whose #sliced API column is 0, as they do not involve any cryptographic computation in the authentication request message, but they are included in our test because their earlier communications involve cryptographic computation. Also, we can notice that there are just a few diffed fields (ranged from 1 to 5) in the request message. Among these diffed fields, 8 apps have one or two system fields (such as timestamp), 20 apps have user input (e.g., username), and 15 apps have cryptographically computed fields in the authentication request message. Meanwhile, all of their response messages are not identical, but 18 of them (78%) only contain system field differences in the response message (some other differences include cookies, etc).

Regarding how long AUTOFORGE takes to test each app server, we note that the most time consuming part is the user registration and the manual user login process. Usually these processes took two to five minutes. The rest of the execution of AUTOFORGE only took less than 10 seconds each to automatically finish password brute-force testing under the setting of N being 20.

C. Leaked Username and Password Probing Testing

The second test we performed is the leaked data probing attack. Being able to generate valid request messages, we would then be able to test whether a leaked username and password exists in the remote mobile service. Through a one time forgery, an attacker can easily find a victim's username and password without performing any brute-force guessing because of the password reuse practice among many users [15], [21].

In the past several years, there were hundreds of millions of leaked passwords and user accounts [7], [31], and such a leaked data probing attack can be easily launched. While the server can limit the origin of the request message (e.g., by limiting a given IP address with only limited number of login attempts, though this is not a good practice as it might cause trouble for some campus networks when a network proxy is used), if an attacker performs distributed testing, such an attack is very challenging to prevent.

To determine whether a service provider is vulnerable to this leaked data probing attack, we performed a simple test. In particular, for ethical reasons, we did not use any of the leaked database accounts, and instead we registered 19 more users in the services we tested (in addition to the two users we registered in password brute-forcing testing). Starting from a single IP address, we keep mutating the the username and wrong password pair in the first 20 request messages, with the 21st request message containing a correct username and password. If the server allows the login, then it means the server is vulnerable to this type of attack. Without any surprise, the server side of all the 76 apps we tested are vulnerable to this leaked data probing attack.

D. Facebook Access Token Hijacking Testing

The third test we performed is to identify the access token hijacking vulnerability in the mobile service. Today, many mobile apps support users logging in to their services with the users' Facebook, Google, Microsoft, or Twitter accounts. For instance, among the tested 76 apps, we found that 36 of them (47%) support Facebook Login, 28 (37%) support Google Login, 5 (7%) support Twitter Login. For a proof-of-concept, we focus on the most popular Facebook Login and demonstrate how to launch an access token hijacking vulnerability test against it. Typically, when a user connects to the app service with Facebook Login, the app will obtain an access token for that particular user and that app, and this token can provide a temporary, secure access to Facebook APIs such as querying user's information stored in Facebook. However, this per-app issued access token is portable, and other apps can use the same user's Facebook token to access the user's private information if the app service does not check the origin of the token. This attack has been described as an access token misuse attack [36] or access token hijacking attack [2].

To perform this test, essentially what we want is to log in to a vulnerable app server by using the Facebook access token that is issued to other apps. Therefore, we just need to substitute an access token (stolen) from other apps, and test whether the app server still allows access and returns a user's private data (again, the fundamental reason is because the app server mistakenly uses the token as authentication [36]). While we could apply our *Message Field Inference* to infer the fields of our interest in the authentication request messages, we notice that many

```
<script type="text/javascript">window.location.href="fbconnect:
\/\/success#granted_scopes=email\u00252Ccontact_email\u00252Cp
ublic_profile&denied_scopes=&access_token=CAAUbRqhb6ggBAEtOE6v
cAjUGqfficRiVUj2WZALM330EBSqDIo98pFEVBgiIhVCgbHihV3qmjgDKr5eDG
BqrhVotkGWQUbaIcXTpxAOHGPskQVLsuJ59PrysHMz6zzAZCx4GAovndOmZAb4
EIXAlLSlvaZCGVyevED2B53FOpAtrPdlaDmh67wKjj56lO7epMtT69ZAXYCQZD
ZD&expires_in=5140807";</script>
```

(a) Facebook Confirmation Message

```
GET /v2.2/me?access_token=CAAUbRqhb6ggBAEtOE6vcAjUGqfficRiVUj2
WZALM330EBSqDIo98pFEVBgiIhVCgbHihV3qmjgDKr5eDGBqrhVotkGWQUbaIc
XTpxAOHGPskQVLsuJ59PrysHMz6zzAZCx4GAovndOmZAb4EIXAlLSlvaZCGVye
vED2B53FOpAtrPdlaDmh67wKjj56lO7epMtT69ZAXYCQZDZD&format=json&s
dk=android HTTP/1.1
x-newrelic-id: XAYCV1ZADgsAUFRTBQ==
User-Agent: FBAndroidSDK.3.20.0
Content-Type: multipart/form-data; boundary=3i2ndDfv2rTHiSisAb
ouNdArYfORhtTPEefj3q2f
Accept-Language: en_US
Host: graph.facebook.com
Connection: Keep-Alive
Accept-Encoding: gzip
```

(b) Client Request Message to Facebook

```
{"id":"109829469364819","email":"testappserver2016\u0040gmail.
com","first_name":"Fndss","gender":"male","last_name":"Lndss",
"link":"https:\/\/www.facebook.com\/app_scoped_user_id\/109829
469364819\/","locale":"en_US","name":"Fndss Lndss","timezone":
-5,"updated_time":"2015-08-17T03:27:04+0000","verified":false}
```

(c) Facebook Response Message

```
POST /api/v1/socials/FACEBOOK/put?timestamp=2015-08-17%2001%3A
16%3A23&sid=0bcd1165dbcc44718b95f35c6ee70fb9&v=1.1&client=andr
oid&accessToken=CAAUbRqhb6ggBAEtOE6vcAjUGqfficRiVUj2WZALM330EB
SqDIo98pFEVBgiIhVCgbHihV3qmjgDKr5eDGBqrhVotkGWQUbaIcXTpxAOHGPs
kQVLsuJ59PrysHMz6zzAZCx4GAovndOmZAb4EIXAlLSlvaZCGVyevED2B53FOp
AtrPdlaDmh67wKjj56lO7epMtT69ZAXYCQZDZD&app_key=A4H0P4JN&langua
ge=en&cv=3.10.0&currency=USD&sign=6992022E02F34E7ED5CD6CF19795
BD86&providerUserId=109829469364819&email=testappserver2016%40
gmail.com HTTP/1.1
x-newrelic-id: XAYCV1ZADgsAUFRTBQ==
User-agent: LightInTheBox 3.10.0(Android; 17; 4.2.2; 480_752;
WIFI; generic; I9100; en)
Host: api.miniinthebox.com
Connection: Keep-Alive
Accept-Encoding: gzip
Content-Type: application/x-www-form-urlencoded
Cookie: AKAMAI_FEO_TEST=B; ASRV=A_201505081100; cookie_test=pl
ease_accept_for_session; JSESSIONID=1qfesxjfnhxas1s1sbde9uut9n
Content-Length: 0
```

(d) Client Authentication Request Message to App Server

Fig. 4. Access Token Hijacking Attack with `miniinthebox` App.

of the fields of our interest can be inferred directly from the response messages sent by Facebook. For instance, as shown in Fig. 4(d), we need to recognize five fields: `timestamp`, `accessToken`, `sign`, `providerUserId`, and `email`. Among them, `accessToken` and `providerUserId` can be inferred directly from the Facebook response message, which is well defined by the Facebook API.

In particular, during the Facebook Login process, Facebook will send a response message as shown in Fig. 4(a) from https://m.facebook.com/v2.2/dialog/oauth/, and we can directly parse this response message to get the `access_token` (because the format is defined by Facebook and every app follows it). Next, a client app will use this token and send a request message to the Facebook server to query for more information about this user; an example of this request message is shown in Fig. 4(b). Next, Facebook will reply to the client with the queried information such as `id`, `email`, `first_name`, etc., about this user. This response message, as shown in Fig. 4(c) also has well-defined fields by Facebook, and we just need to parse them to retrieve the information of our interest such as the `id` field. We can notice from Fig. 4(d) that `id`, `access_token`,

TABLE IV. THE DETAILED RESULT ON THE SECURITY TOKEN SUBSTITUTION TESTING

App Package Name	Step ❶	Step ❷				Step ❸							Step ❹		Step ❺		
	#Input Msg	#Traced API	Encryption?	Hashing?	Signing?	#DiffedField	#SysField	#InputField	#CryptoField	Access Token?	ID?	Email?	EqualResonse?	SysField Only?	#Sliced API	#Request	Vulnerable?
anews.com	2	144	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✗	0	1	✗
com.ad60.songza	2	185	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✗
com.askfm	2	790	✗	✗	✓	2	0	1	1	✓	✗	✗	✗	✓	7	1	✗
com.biggu.shopsavvy	2	611	✗	✓	✗	2	0	1	1	✓	✗	✗	✗	✓	7	1	✓
com.bukalapak.android	2	521	✗	✗	✗	2	0	2	0	✓	✓	✗	✗	✓	0	1	✓
com.careerjet.android	2	231	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✓
com.clearchannel.iheartradio.controller	2	800	✗	✗	✗	1	0	1	0	✗	✓	✗	✗	✗	0	1	✗
com.dictionary.flashcards	2	72	✗	✗	✗	2	0	2	0	✓	✓	✓	✗	✓	0	1	✗
com.espn.score_center	2	567	✗	✗	✗	2	0	2	0	✓	✓	✓	✗	✗	0	1	✗
com.expedia.bookings	2	1090	✗	✗	✗	2	0	2	0	✓	✓	✗	✗	✗	0	1	✗
com.geeksoft.wps	2	364	✗	✓	✗	2	0	1	1	✗	✓	✗	✗	✓	7	1	✗
com.imdb.mobile	2	947	✗	✗	✓	3	1	1	1	✓	✗	✗	✗	✓	7	1	✗
com.jabong.android	2	719	✗	✗	✗	2	0	2	0	✓	✓	✓	✗	✓	0	1	✗
com.mediafire.android	2	858	✗	✓	✗	2	0	1	1	✓	✗	✗	✗	✓	8	1	✓
com.meucarrinho	2	332	✗	✓	✗	4	2	1	1	✓	✗	✗	✗	✓	7	1	✓
com.miniinthebox.android	2	572	✗	✓	✗	5	2	2	1	✓	✓	✓	✗	✓	7	1	✓
com.mobilesrepublic.appygamer	2	204	✗	✗	✗	1	0	1	0	✗	✓	✓	✗	✓	0	1	✗
com.mobilesrepublic.appygeek	2	929	✗	✗	✗	1	0	1	0	✗	✓	✓	✗	✓	0	1	✗
com.myfitnesspal.android	2	958	✗	✗	✗	2	0	2	0	✓	✓	✗	✗	✓	0	1	✗
com.noom.walk	2	316	✗	✗	✗	2	0	2	0	✗	✓	✓	✗	✗	0	1	✗
com.picsart.studio	2	2622	✗	✗	✗	4	0	4	0	✓	✓	✓	✗	✓	0	1	✗
com.rebtel.android	2	421	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✗
com.skout.android	2	583	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✗
com.slacker.radio	2	529	✗	✗	✗	2	0	2	0	✓	✓	✗	✗	✗	0	1	✗
com.somcloud.somnote	2	74	✗	✗	✗	3	0	3	0	✗	✓	✓	✗	✓	0	1	✓
com.soundcloud.android	2	415	✗	✗	✗	2	0	2	0	✓	✗	✗	✗	✓	0	1	✗
com.stuckpixelinc.funnypics	2	243	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✓
com.textmeinc.textme	2	34	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✗
com.zillow.android.zillowmap	2	921	✗	✗	✗	2	0	2	0	✓	✓	✗	✗	✓	0	1	✓
taxi.android.client	2	490	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✓	0	1	✗
wp.wpbeta	2	202	✗	✗	✗	1	0	1	0	✓	✗	✗	✗	✗	0	1	✗

and `email` have been used in the authentication request message even though the client app (our running example `miniinthebox`) uses different names for some of the fields. For `timestamp` and `sign` fields, we will still rely on our `Message Field Inference` to identify them.

We tested whether these 76 app servers in §IV-B are vulnerable to this access token hijacking attack. While we have found 36 of them that use Facebook Login, in fact 5 apps were actually buggy in this feature (and we cannot launch the Facebook Login for them). Therefore, we only have 31 apps that were tested. The test is slightly different compared to our password brute force test in that we only need to register one user on Facebook (with the `testappserver2016@gmail.com` account). After that, we need to intercept the Facebook access token `oauth` confirmation message as shown in Fig. 4(a), and the Facebook user information query message as shown in Fig. 4(c), from which we extract the fields of our interest such as `access_token` and `id`. Next, we send two authentication request messages to the app server, and apply the message diffing to identify other fields. After that, we substitute the `access_token` and `id` field in the client authentication request message, and replay the execution of the cryptographically computed fields such as `sign` to test whether the server is vulnerable or not.

The detailed result of the tested 31 apps is presented in Table IV. Most columns share the same meaning as in Table III, except we added whether the request messages use Access Token, ID, or Email from the 12th to 14th column. We can notice from Table IV that 21 (68%) of the apps use HTTPS, and we only need to send two authentication request messages. Interestingly, only 7 out of 31 (23%) of the request messages involves hashing or signing. Also, we notice not all the request messages use the access token, and some of them use the ID returned from Facebook for the authentication. Meanwhile, all the response messages for the same user's login are not identical, but the major difference still comes from the timestamp field. Finally, we only send one request message to the server and we only find 9 out of 31 (29%) apps that are vulnerable to the Facebook token hijacking attack.

V. DISCUSSIONS

A. Security Implications

AUTOFORGE has demonstrated that lack of security checks at the server side can lead to several severe attacks such as password brute forcing, leaked username and password probing, and access token hijacking. This is a very serious problem considering that a large volume of popular apps, including CNN, Expedia, iHeartRadio, and Walmart as confirmed in our experiment are vulnerable to these attacks. While it is true that an adversary cannot sniff the password because of HTTPS, an attacker can launch a malicious login attack in an owned device to install self-signed certificates and automatically forge the request messages even though there are cryptographic constraints. As such, we would like to raise awareness for app developers: only using HTTPS cannot defeat password brute-forcing, and neither can hashing and (one-way) signing of client request messages.

Therefore, we need to examine the techniques that can be used by app developers to mitigate or prevent the automatic forgery of user request messages, especially in the scenario of user authentication, and they can be summarized as follows:

- **Limiting the number of login attempts**. One simple solution app developers can adopt is to keep a login attempt state at the server side and limit the number of login attempts within a certain time window. We only found 11 out of 76 apps (14%), such as `com.imdb.mobile`, that followed this approach. While this solution cannot defeat leaked username and password probing attacks, it can defeat at least user password brute forcing. Meanwhile, unlike CAPTCHA and two factor-authentication discussed below, this defense will not change any user's experience.
- **Using CAPTCHA**. Automatic data forgery is not a new attack, and there are already solutions to mitigate this. One way that has been widely used on the desktop is the CAPTCHA [34]. A CAPTCHA is a program that protects websites against automated resource abusing or login attempts. However, we have not seen much usage in mobile apps. We believe one reason is that CAPTCHA might hurt user experience. However, as we have demonstrated in this paper, to really slow down attackers, CAPTCHA is a viable approach, though CAPTCHA can also be broken [33].
- **Two-factor authentication**. Another intuitive way to slow down the forgery of user request messages (including the authentication) is to adopt two-factor authentication [38]. Similar to CAPTCHA, it will certainly hurt user experience, but it is unlikely for attackers to successfully compromise two channels.
- **Two-way authentication**. The most effective way to prevent client side data forgery is to authenticate the client as well using a two-way (i.e., mutual) authentication [16]. Two-way SSL is one such an example, and it uses digital signatures to authenticate both the server and the client with their corresponding certificates. However, it requires an extra effort of client certificate exchange and imposes additional complexity and cost. Therefore, we have not observed any apps that use this technique.

B. Limitations and Future Work

While we have made a first step demonstrating the feasibility of automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services, there are a number of avenues for future improvement. In the following, we discuss the limitations of AUTOFORGE and outline future work.

First, AUTOFORGE currently only focuses on HTTP/HTTPS protocols. There are certainly apps that use other protocols such as proprietary non-plaintext protocols. While our global optimal sequence alignment algorithm (i.e., the Needleman-Wunsch algorithm [27]) might be able to align the two diffed messages to identify the diffed fields for non-plaintext protocols, we have not evaluated it yet. Our next step is to test how AUTOFORGE would perform with non-plaintext protocols.

Second, AUTOFORGE only performs lightweight API level tracing of app's execution, and assumes user input (such as the entered username) would not be transformed (recall we use content patching to identify the direct user inputs). However, a user entered input could be translated into other forms. To really track the possible transformations of the user input, a better way is to perform fine-grained instruction level data flow tracking. Therefore, we plan to integrate a taint analysis engine such as TaintDroid [19] into AUTOFORGE to track the user's input such that we can still recognize the input in the request messages.

Third, AUTOFORGE currently only deals with the cryptographic APIs listed in Table I. If an app uses other APIs or native code, AUTOFORGE has to include them. We plan to examine more apps and enrich the list with more APIs if there are any. Meanwhile, if an app uses its own private cryptographic functions, AUTOFORGE has to perform additional analysis (such as those mentioned in Dispatcher [9], Aligot [11], or the methods described by Grobert et al. [20]) to recognize these functions.

Fourth, our security test might have false positives because of the limited number of tests we performed. For instance, an app service could block the user after the $(N+1)$-th failure without us detecting it (because of our threshold of maximum N guesses), and we would have to enlarge N to prune this. Note that we set the parameter N to small numbers just for ethical considerations, and a real attack would not be constrained by this.

Finally, AUTOFORGE will enable many other security tests, such as SQL injection by manipulating the corresponding request fields (e.g., we can append certain data to the username). In fact, we did find one app that is vulnerable to SQL injection among the 76 apps. We leave the large scale systematic study of this type of vulnerability to our future work.

C. Ethics

The goal of designing AUTOFORGE is to apply it to find vulnerabilities at the server side. In this case, we have to inevitably send unnecessary packets to the service providers. We do take ethics into consideration by minimizing the number of messages sent to the server (recall the maximum number of messages we sent is $N+1$). Also, we have made responsible disclosure and notified all the vulnerable app vendors. In fact, shortly after we reported the vulnerabilities, three vendors patched their services by only allowing a limited number of failed logins. For instance, the iHeartRadio app has limited the maximum number of login attempts to 15, the ESPN score center app limits it to 3, and the Slacker Radio app limits it to 6. We believe many other vendors will also patch their services very soon.

VI. RELATED WORK

At a high level, our work is related to protocol reverse engineering, application dialogue replay, password brute forcing, and mobile app vulnerability discovery. In this section, we review these works and compare AUTOFORGE with them.

Protocol Reverse Engineering. There is a large body of research focusing on protocol reverse engineering. Earlier efforts (e.g., [8], [12], [24]) inferred the protocol format from network traces. Protocol informatics [8] used the Needleman-Wunsch algorithm [27] to align the protocol messages and infer the protocol format. Discoverer [12] proposed tokenization, recursive clustering, and merging techniques to handle both text and binary protocols from network traces.

Instead of only using the network traces, the other direction of protocol reverse engineering is to use dynamic binary analysis

(taint analysis in particular) to reveal the protocol formats. A number of systems or tools (e.g., [9], [10], [14], [25], [39]) have been proposed. Among them, Polyglot [10] made the first attempt of using binary code analysis to infer the protocol formats, Tupni [14] recovers more fine-grained protocol formats, and Dispatcher [9] focused on encrypted protocol message reverse engineering. We plan to apply the techniques proposed by these efforts to recover the Android apps' protocol in a more general way such as also inferring binary data based protocols.

Application Dialogue Replay. AUTOFORGE employs cryptographic function replay to generate the authenticated messages, which is similar to the existing application dialogue replay systems. Similar to protocol reverse engineering, there are also two categories of techniques: purely network traces based, and binary code analysis based.

Similar to Protocol Informatics [8], RolePlayer [13] aligns the byte-wise sequences of the protocol messages from network traces, and then identifies and mutates some specific fields for the application dialogue replay. By leveraging binary code analysis, Replayer [28] enables more automatic replay. While AUTOFORGE appears to be quite similar to these replay systems, none of the existing efforts focused on cryptographic protocol fields mutation (RolePlayer assumed there is no such field in the protocol message, and Replayer set cryptographic fields in its future work), which is the exact focus of AUTOFORGE.

Password Brute Forcing. Password based authentication has been the de facto standard to protect access to sensitive information, with no exceptions to mobile apps and services. It has always been a major focus for attackers over years, and there are many efficient and practical ways of brute force cracking a user's password. For instance, assuming access to the password file, attackers can use a dictionary based attack to break user passwords. Recently, there were also significant efforts to make dictionary attacks smarter by employing Markov models (e.g., [26]), probabilistic context free grammars (e.g., [37]), and history based guessing (e.g., [40]). There are also approaches to make the password brute forcing much faster. Using rainbow tables is one such approach, which consists of massive tables of pre-calculated hashes, trading increased memory storage for reduced computation time [29]. While AUTOFORGE does focus on password brute forcing, it shows the new context of brute forcing user passwords for mobile apps with the techniques of automatically generating mutated passwords in the authenticated request message.

Mobile App Vulnerability Discovery. In the past several years, a considerable amount of efforts have focused on discovering various vulnerabilities in mobile apps. For instance, TaintDroid [18] detects privacy leakage vulnerabilities by tracking information flows. PiOS [17] uses static analysis to detect such leaks in iOS apps. CHEX [23] detects component hajacking vulnerabilities in Android apps by using a data-flow based static analysis approach. SMV-Hunter [32] detects man-in-the-middle SSL/TLS vulnerabilities with a hybrid static and dynamic analysis. However, few efforts have been focusing on identifying the vulnerabilities in an app's server side. AUTOFORGE made such a step in this direction and demonstrated that there are also serious security vulnerabilities such as password brute forcing if app server developers do not perform the necessary security checks.

VII. CONCLUSION

We have presented AUTOFORGE, a tool that can automatically forge cryptographically consistent messages from the client side to test whether the server side of an app contains security vulnerabilities such as brute-forcing, leaked username and password probing, and access token hijacking. To enable our security test, we have developed a set of techniques to automatically infer protocol fields, label response messages, replay cryptographic function execution, and regenerate request messages. Our experimental results show that among the 76 tested popular apps (each with millions of installs), 65 of their servers (86%) are vulnerable to password brute forcing attacks, all of them (100%) are vulnerable to leaked username and password probing attacks, and 9 of them (12%) are vulnerable to Facebook access token hijacking attacks. We have performed responsible disclosure and notified each vulnerable app vendor, and three of the service providers, including ESPN and iHeartRadio, have patched their services shortly after our notification.

ACKNOWLEDGMENT

We are grateful to our shepherd Christopher Kruegel, and the anonymous reviewers for their extremely helpful feedback. We also would like to thank Erick Bauman and Murat Kantarcioglu for proof-reading of the paper. This work was partially supported by The Air Force Office of Scientific Research (AFOSR) under Award No. FA-9550-12-1-0077. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR.

REFERENCES

[1] "Burp suite," https://portswigger.net/burp/.

[2] "Facebook token hijacking," https://developers.facebook.com/docs/facebook-login/security/#tokenhijacking.

[3] "Genymotion," https://www.genymotion.com/.

[4] "Statistics and facts about app stores," http://www.statista.com/topics/1729/app-stores/.

[5] "Ui/application exerciser monkey," https://developer.android.com/tools/help/monkey.html.

[6] "Xposed module repository," http://repo.xposed.info/.

[7] "Hackers released the passwords of over 70 million chinese internet accounts," https://dazzlepod.com/rootkit/, 2011.

[8] M. Beddoe, "The protocol informatics project," http://www.4tphi.net/~awalters/PI/PI.html.

[9] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *CCS*, Chicago, Illinois, USA, 2009, pp. 621–634.

[10] J. Caballero and D. Song, "Polyglot: Automatic extraction of protocol format using dynamic binary analysis," in *CCS*, Alexandria, Virginia, USA, 2007, pp. 317–329.

[11] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *CCS*. ACM, 2012, pp. 169–182.

[12] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *USENIX Security Symposium*, Boston, MA, August 2007.

[13] W. Cui, V. Paxson, N. Weaver, and R. H. Katz, "Protocol-independent adaptive replay of application dialog," in *NDSS*, San Diego, CA, February 2006.

[14] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *CCS*, Alexandria, Virginia, USA, October 2008, pp. 391–402.

[15] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The Tangled Web of Password Reuse," in *NDSS*, February 2014.

[16] W. Diffie, P. C. Van Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes and cryptography*, vol. 2, no. 2, pp. 107–125, 1992.

[17] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *NDSS*, 2011.

[18] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[20] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs." in *RAID*, vol. 6961. Springer, 2011, pp. 41–60.

[21] B. Ives, K. R. Walsh, and H. Schneider, "The domino effect of password reuse," *Commun. ACM*, vol. 47, no. 4, pp. 75–78, Apr. 2004. [Online]. Available: http://doi.acm.org/10.1145/975817.975820

[22] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *NDSS*, San Diego, CA, February 2008.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *CCS*. ACM, 2012, pp. 229–240.

[24] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker, "Unexpected means of protocol inference," in *IMC*. Rio de Janeriro, Brazil: ACM Press, 2006, pp. 313–326.

[25] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," in *IEEE Symposium on Security & Privacy*, Oakland, CA, 2009, pp. 110–125.

[26] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *CCS*, ACM, 2005, pp. 364–372

[27] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[28] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *CCS*, 2006.

[29] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology-CRYPTO 2003*. Springer, 2003, pp. 617–630.

[30] B. Schneier, "Cryptography: The importance of not being different," *Computer*, vol. 32, no. 3, pp. 108–109,112, Mar. 1999.

[31] M. Siegler, "One of the 32 million with a rockyou account? you may want to change all your passwords. like now," http://techcrunch.com/2009/12/14/rockyou-hacked/, 2009.

[32] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *NDSS*, San Diego, CA, February 2014.

[33] J. Tam, J. Simsa, S. Hyde, and L. V. Ahn, "Breaking audio captchas," in *NIPS*, 2008, pp. 1625–1632.

[34] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *Advances in Cryptology — EUROCRYPT 2003*. Springer, 2003, pp. 294–311.

[35] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[36] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization." in *USENIX Security*, 2013, pp. 399–314.

[37] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *SP*, 2009, pp. 391–405.

[38] K. P. Weiss, "Method and apparatus for positively identifying an individual," Jan. 19 1988, uS Patent 4,720,860.

[39] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in *NDSS*, San Diego, CA, February 2008.

[40] Y. Zhang, F. Monrose, and M. K. Reiter, "The security of modern password expiration: An algorithmic framework and empirical analysis," in *CCS*, ACM, 2010, pp. 176–186.

APPENDIX

In §IV-B, we presented the detailed experimental results for 23 app servers, and these apps are selected based on their categories. The detailed app classification, their version, and protocol information is presented in Table V. The result for the 53 other app servers is presented in Table VI. The column of Table VI shares the same format as Table III. Detailed explanation of these results is elided for brevity.

TABLE V. THE CATEGORY, INSTALLS, APP NAME, VERSION, AND PROTOCOL INFORMATION FOR THE TESTED 76 APPS.

Category	#install	App Package Name	Version	Protocol
Books & Reference	100,000,000	com.sirma.mobile.bible.android	6.0.3	HTTPS
Books & Reference	50,000,000	com.kobobooks.android	6.3.13738	HTTPS
Books & Reference	5,000,000	com.overdrive.mobile.android.mediaconsole	3.4.0	HTTPS
Books & Reference	5,000,000	wp.wpbeta	6.1.0.8	HTTPS
Business	10,000,000	com.sahibinden	2.4.0	HTTPS
Business	5,000,000	com.timesgroup.magicbricks	6.1.2	HTTP
Business	5,000,000	naukriApp.appModules.login	6.3.1	HTTPS
Business	1,000,000	com.careerjet.android	5.1.3	HTTP
Casual	500,000,000	me.pou.app	1.4.67	HTTP
Comics	5,000,000	jp.ebookjapan.ebireader	2.3.79.0	HTTPS
Communication	50,000,000	com.browan.freeppmobile.android	FIAD.BRO.3.7.0.445	HTTP
Communication	50,000,000	com.mx.browser	4.5.0.2000	HTTPS
Communication	50,000,000	com.textmeinc.textme	2.8.8	HTTPS
Communication	50,000,000	ru.mail.mailapp	3.1.2.11965	HTTPS
Communication	10,000,000	com.my.mail	3.1.3.12222	HTTPS
Communication	5,000,000	com.mx.browser.tablet	4.3.5.2000	HTTPS
Communication	5,000,000	com.rebtel.android	3.11.0	HTTPS
Education	5,000,000	com.dictionary.flashcards	1	HTTP
Entertainment	100,000,000	com.imdb.mobile	5.5.6.105561200	HTTPS
Entertainment	50,000,000	com.cgv.android.movieapp	4.0.7	HTTPS
Entertainment	50,000,000	com.dailymotion.dailymotion	4760	HTTPS
Entertainment	10,000,000	com.viewster.androidapp	4.6.3	HTTPS
Entertainment	5,000,000	com.gamefly.android.gamecenter	3.49	HTTPS
Entertainment	5,000,000	com.stuckpixelinc.funnypics	3.3.1	HTTP
Finance	5,000,000	com.netgate	8.22	HTTPS
Health & Fitness	50,000,000	com.fatsecret.android	4.1.2.2	HTTP
Health & Fitness	50,000,000	com.myfitnesspal.android	4.6.1	HTTPS
Health & Fitness	10,000,000	com.noom.walk	1.1.3	HTTP
Lifestyle	50,000,000	com.cookpad.android.activities	5.2.1.0	HTTPS
Lifestyle	50,000,000	com.zillow.android.zillowmap	6.6.8.4011	HTTPS
Lifestyle	10,000,000	com.dominospizza	2.7.0	HTTPS
Lifestyle	5,000,000	cn.etouch.ecalendar2	6.1.5	HTTPS
Media & Video	10,000,000	com.youku.phone	4.7.1	HTTP
Media & Video	5,000,000	com.qiyi.video.market	6.5.1	HTTPS
Media & Video	5,000,000	com.sohu.sohuvideo	4.3.5	HTTP
Media & Video	1,000,000	tv.danmaku.bili	4.2.3	HTTPS
Medical	5,000,000	com.aranoah.healthkart.plus	7.1.6	HTTP
Medical	5,000,000	com.sigmaphone.topmedfree	5.8.1	HTTPS
Medical	5,000,000	leafly.android	2.5.0	HTTP
Music & Audio	100,000,000	com.slacker.radio	6.0.1816	HTTPS
Music & Audio	100,000,000	com.soundcloud.android	15.08.14-release	HTTPS
Music & Audio	50,000,000	com.clearchannel.iheartradio.controller	5.8.0	HTTPS
Music & Audio	10,000,000	com.ad60.songza	5.2.0.0	HTTPS
Music & Audio	10,000,000	com.kugou.android	7.6.1	HTTP
Music & Audio	10,000,000	uk.co.sevendigital.android	5.70.8	HTTPS
News & Magazines	50,000,000	com.cnn.mobile.android.phone	2.8.2	HTTPS
News & Magazines	10,000,000	com.ideashower.readitlater.pro	5.8.5	HTTPS
News & Magazines	5,000,000	anews.com	2.7.166	HTTP
News & Magazines	5,000,000	com.mobilesrepublic.appygamer	5.1.4	HTTP
News & Magazines	5,000,000	com.mobilesrepublic.appygeek	5.1.3	HTTP
Photography	500,000,000	com.picsart.studio	5.6.3	HTTPS
Productivity	50,000,000	com.autodesk.autocadws	3.1	HTTPS
Productivity	50,000,000	com.ecareme.asuswebstorage	2.2.7.8664	HTTPS
Productivity	5,000,000	com.mediafire.android	3.2.3	HTTPS
Productivity	5,000,000	com.somcloud.somnote	2.2.1	HTTPS
Productivity	1,000,000	com.geeksoft.wps	3.0.7	HTTP
Shopping	50,000,000	com.biggu.shopsavvy	9.3.3	HTTPS
Shopping	50,000,000	com.walmart.android	2.8.2	HTTPS
Shopping	10,000,000	com.jabong.android	2.4.1	HTTPS
Shopping	5,000,000	com.bukalapak.android	3.0.1	HTTPS
Shopping	5,000,000	com.meucarrinho	5.6.1	HTTP
Shopping	5,000,000	com.miniinthebox.android	3.10.0	HTTP
Social	100,000,000	com.tumblr	3.9.0.50	HTTPS
Social	50,000,000	com.askfm	2.2.1	HTTPS
Social	50,000,000	com.chatous.pointblank	3.5.1	HTTPS
Social	50,000,000	com.skout.android	4.14.4	HTTP
Social	50,000,000	com.unearby.sayhi	4.39	HTTP
Social	10,000,000	com.match.android.matchmobile	3.2.0	HTTPS
Social	5,000,000	com.tenthbit.juliet	1.8.0	HTTPS
Sports	50,000,000	com.espn.score_center	4.4.1.1	HTTPS
Tools	10,000,000	com.sohu.inputmethod.sogou	7.6	HTTPS
Tools	5,000,000	xcxin.fehd	2.3.0	HTTPS
Transportation	5,000,000	taxi.android.client	5.4.5	HTTPS
Travel & Local	50,000,000	com.expedia.bookings	6.3.1	HTTPS
Travel & Local	5,000,000	com.viamichelin.android.michelintraffic	4.3.0.4	HTTP
Weather	1,000,000	disasterAlert.PDC	3.2	HTTPS

TABLE VI. THE DETAILED PASSWORD BRUTE-FORCING TESTING RESULT FOR THE OTHER 53 APP SERVERS.

		Step ❶	Step ❷				Step ❸				Step ❹		Step ❺		
Category	**App Package Name**	**#Input Msg**	**#Traced API**	**Encryption?**	**Hashing?**	**Signing?**	**#DiffedField**	**#SysField**	**#InputField**	**#CryptoField**	**EqualResonse?**	**SysField Only?**	**#Sliced API**	**#Request**	**Vulnerable?**
Books & Reference	com.kobobooks.android	4	240	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Books & Reference	com.overdrive.mobile.android.mediaconsole	4	448	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Books & Reference	wp.wpbeta	4	333	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Business	com.careerjet.android	4	28	✓	✗	✗	2	1	0	1	✗	✓	9	21	✓
Business	com.timesgroup.magicbricks	4	89	✗	✗	✓	2	0	0	2	✗	✓	20	21	✓
Business	naukriApp.appModules.login	4	115	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Communication	com.mx.browser	4	195	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Communication	com.mx.browser.tablet	4	178	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Communication	com.my.mail	4	340	✗	✓	✗	3	0	2	1	✗	✓	7	21	✓
Communication	com.rebtel.android	4	208	✗	✓	✗	5	2	2	1	✗	✓	8	21	✗
Communication	com.textmeinc.textme	4	241	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Communication	ru.mail.mailapp	4	83	✗	✓	✗	3	0	2	1	✗	✓	7	21	✓
Entertainment	com.cgv.android.movieapp	4	67	✗	✓	✗	3	0	1	2	✗	✓	18	21	✓
Entertainment	com.dailymotion.dailymotion	4	34	✗	✗	✓	4	1	2	1	✗	✓	12	21	✓
Entertainment	com.gamefly.android.gamecenter	4	86	✗	✗	✓	4	1	2	1	✗	✓	7	21	✓
Entertainment	com.stuckpixelinc.funnypics	4	31	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Entertainment	com.viewster.androidapp	4	626	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Health & Fitness	com.myfitnesspal.android	4	269	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Health & Fitness	com.noom.walk	4	48	✗	✓	✗	3	0	2	1	✗	✗	18	21	✓
Lifestyle	cn.etouch.ecalendar2	4	1232	✓	✗	✗	1	0	0	1	✗	✓	11	21	✓
Lifestyle	com.dominospizza	4	265	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Lifestyle	com.zillow.android.zillowmap	4	242	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Media & Video	com.qiyi.video.market	4	1169	✗	✓	✗	4	1	2	1	✗	✓	18	3	✗
Media & Video	com.sohu.sohuvideo	4	72	✗	✓	✗	2	0	1	1	✗	✓	7	10	✗
Media & Video	tv.danmaku.bili	4	1294	✓	✓	✗	3	0	1	2	✗	✓	15	3	✗
Medical	com.sigmaphone.topmedfree	4	49	✓	✗	✗	1	0	0	1	✗	✓	1	15	✗
Medical	leafly.android	4	38	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Music & Audio	com.ad60.songza	4	132	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Music & Audio	com.clearchannel.iheartradio.controller	4	1237	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Music & Audio	com.kugou.android	4	637	✓	✓	✗	4	1	1	2	✗	✓	22	21	✓
Music & Audio	com.soundcloud.android	4	60	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Music & Audio	uk.co.sevendigital.android	4	1792	✗	✗	✓	5	2	2	1	✗	✓	7	21	✓
News & Magazines	anews.com	4	192	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
News & Magazines	com.ideashower.readitlater.pro	4	239	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
News & Magazines	com.mobilesrepublic.appygamer	4	276	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
News & Magazines	com.mobilesrepublic.appygeek	4	883	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Productivity	com.ecareme.asuswebstorage	4	85	✗	✓	✓	6	3	1	2	✗	✓	17	21	✓
Productivity	com.geeksoft.wps	4	25	✗	✓	✗	3	0	2	1	✗	✓	7	21	✓
Productivity	com.mediafire.android	4	201	✗	✓	✗	3	0	2	1	✗	✓	8	12	✗
Productivity	com.somcloud.somnote	4	743	✗	✓	✓	5	2	1	2	✗	✓	14	21	✓
Shopping	com.bukalapak.android	4	430	✓	✗	✗	1	0	0	1	✗	✓	1	21	✓
Shopping	com.jabong.android	4	780	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Shopping	com.meucarrinho	4	138	✗	✓	✗	5	2	2	1	✗	✓	7	21	✓
Shopping	com.miniinthebox.android	4	228	✓	✓	✗	4	1	1	2	✗	✓	19	21	✓
Shopping	com.walmart.android	4	343	✗	✗	✗	2	0	2	0	✗	✗	0	21	✓
Social	com.askfm	4	75	✗	✗	✓	3	0	1	2	✗	✓	7	21	✓
Social	com.chatous.pointblank	4	43	✗	✗	✗	1	0	0	0	✗	✓	1	21	✓
Social	com.match.android.matchmobile	4	308	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Social	com.skout.android	4	115	✗	✓	✗	3	0	2	1	✗	✓	7	3	✗
Social	com.tenthbit.juliet	4	24	✗	✗	✗	2	0	2	0	✗	✓	0	21	✓
Social	com.unearby.sayhi	4	60	✗	✓	✗	2	0	1	1	✗	✓	7	21	✓
Tools	xcxin.fehd	4	73	✓	✗	✗	2	0	1	1	✗	✓	7	21	✓
Travel & Local	com.viamichelin.android.michelintraffic	4	33	✗	✗	✗	3	0	3	0	✗	✓	0	21	✓